Optimized dynamic compilation of dataflow representations for multimedia applications

Jérôme Gorin, Mickaël Raulet, Françoise Préteux

Springer manuscript No.
(will be inserted by the editor)

Optimized dynamic compilation of dataflow representations for multimedia applications

Jérôme Gorin · Mickaël Raulet · Françoise Prêteux

Received: date / Accepted: date

Abstract This paper proposes two optimization methods based on dataflow representations and dynamic compilation that enhance flexibility and performance of multimedia applications. These optimization methods are intended to be used in an adaptive decoding context, or, in other terms, where decoders have the ability to adapt their decoding processes according to a bitstream. This adaptation is made possible by coupling the decoding information to process a stream inside a coded stream.

In this paper, we use dataflow representations from the upcoming MPEG Reconfigurable Media Coding (RMC) standard to supply the decoding information to adaptive decoders. The benefits claimed by MPEG RMC are a reuse of coding tools between different specifications of decoder and an execution scalability on different processing units with a single specification, which can target either hardware and/or software platforms. These benefits are not yet achievable in practice as these specifications are not used at the receiver side in MPEG RMC. We valid these benefits and propose two optimizations for the generation and the execution of dataflow models: the first optimization takes benefits of the reuse of coding tools to reduce the time to obtain – *configure* – enforceable decoders. The second provides an efficient, dynamic and scalable execution according to the features of the execution platform. We show the practical impact of these two optimizations on two decoder representations compliant with the MPEG-4 part 2 *Simple Profile* standard and the MPEG-4 *Advanced Video Coding* standard. The results shows that configuration time can be reduced by 3 and the performance of decoders can be increased by 50%.

Keywords MPEG Reconfigurable Media Coding · MPEG Reconfigurable Video Coding · MPEG Reconfigurable Graphic Coding · dynamic compilation · adaptive decoding · multimedia application · dataflow program · reconfiguration · scalable execution · Dataflow Process Network · Dataflow scheduling

Jérôme Gorin
ARTEMIS, Institut Télécom SudParis, UMR 8145, Evry, France
E-mail: Jerome.Gorin@it-sudparis.eu

Françoise Prêteux
MINES ParisTech, 60 Boulevard Saint-Michel, 75272 Paris, France
E-mail: Francoise.Preteux@minesparistech.fr

Mickaël Raulet
IETR, INSA Rennes, F-35043, Rennes, France
E-mail: mickael.raulet@insa-rennes.fr

1 Introduction

The presence of multimedia has changed significantly over the last two decades. The increasing popularity of digital communication (e.g., Digital Terrestrial Television) and multimedia terminals (e.g., smartphones) has brought the use of multimedia standards to a large number of customers. Multimedia applications are dominated by international "fixed" standards such as MPEG-2, MPEG-4 and VC-1. But new competitors are entering into this highly competitive market; these competitors look to new coding standards able to contain more multimedia content in less data. These players develop their own technologies in such a way that technology providers have to support and follow these technologies so as not to fall too quickly into obsolescence. The traditional approach to multimedia processing curbs the evolution of multimedia applications, as it induces a long delay between the development of a new technology and its implementation in a device.

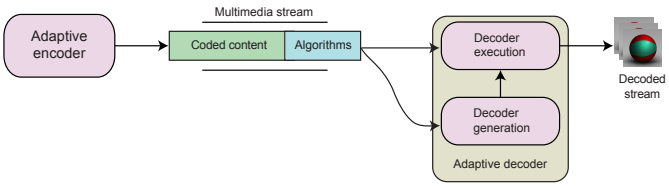


Fig. 1 Adaptive decoding environment.

The concept of adaptive decoding is a novel approach to compression in which a multimedia decoder is dynamically constructed on a device according to a coded stream [43]. It has been primarily addressed to tackle the limitations of interoperability and evolution inherent in the use of static, fixed approaches of video decoding. To this end, an adaptive decoding environment (Fig. 1) is composed of an adaptive encoder, which transmits the compressed multimedia content, and an adaptive decoder, that processes the multimedia content on the receiver side. In this network, one transmitter not only provides the compressed multimedia content, it also provides a generic representation of the algorithms required to decode it, called *configuration information*. The advantages of this new approach are twofold. First, it reduces times for proposing, standardizing and deploying new video coding concepts. Secondly, it avoids proprietary implementation of decoders in devices, an undeniable source of inconsistency between multimedia applications.

The adaptive decoder in this environment is similar to a virtual machine, it dynamically compiles or interprets configuration information from one multimedia application in order to process a stream. However, the decrease of performance induced by dynamic compilation becomes a significant issue when dealing with multimedia processing. Indeed, a decoding process is computation intensive and it typically requires full exploitation of the processing units, such as Digital Signal Processor (DSP) or Graphics Processing Unit (GPU), available in the execution system at the receiver side. Conventional languages, such as those used in virtual machines, are usually sequential, *ergo* they can hardly scale heterogeneous resources. In this context, the abstraction of configuration information becomes a critical property to exploit the widest range of systems capable of receiving video content.

This paper extends the work investigated in [19] where dataflow representations are used to describe decoder algorithms for adaptive decoders. A dataflow representation is a paradigm of signal processing, where an application is composed of a graph with vertices, which represent operations of an application, and edges, which represent the data flowing between operations. In a multimedia context, the MPEG Reconfigurable Media Coding (RMC) standards offer a standardized approach to dataflow representations to ensure the permanence of the model. These standards also provide decoder specifications in dataflow representation form; these are modular, portable and user-friendly specification of decoder in a unified form. In MPEG RMC operations of dataflow representation are used to describe coding tools within MPEG standards. The benefits of dataflow representations are (1) coding tools from dataflow operations can be reused across specification of decoder (2) the strong encapsulation of coding tools provides an explicit representation of the potential concurrency between the algorithms of a decoder. MPEG RMC also provides synthesis tools for decoder developers to derive specifications into C, Java or HDL code. As such, MPEG RVC has already proved its benefits to provide statically compiled decoders fitted to a wide range of platforms [24].

We extend the use of MPEG RMC specifications to the adaptive decoding context and we propose two optimization algorithms that reduce both the synthesis process and the degraded performance caused by dynamic compilation:

- The first algorithm reduces the synthesis process by compiling coding tools never used before.
- The second algorithm detects, regroups and optimizes specific coding tools to be efficiently executed on sequential processors.

The paper is organized as follows. We first give a brief introduction to the entry-level mechanisms that form a dataflow representation and to the benefit of dataflow representation in an adaptive decoding context. We then review the state-of-art of techniques used to optimize the configuration and the execution of dataflow representation. We finally detail the proposed two optimizations in two dedicated sections. We conclude this paper by introducing one implementation of these two optimizations in an adaptive open-source decoder, namely the *Just-In-Time Adaptive Decoder Engine* (JADE). The multimedia applications used are from international coding standards, the MPEG-4 part 2 and the MPEG-4 part 10 Advanced Video Coding (AVC). The results show that the proposed optimization can speed up by a factor of 3 configuration time of multimedia applications and can increase by of factor of 1.5 the performance of the same applications in comparison with conventional approaches.

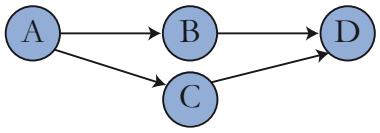


Fig. 2 A dataflow representation with 4 operations (A, B, C, D) and 4 edges.

2 Describing decoder algorithms: the dataflow approach

The adaptive decoding environment for video decoders was first conceived in [43]. In this environment, configuration information for the adaptive decoder was based on a specific imperative syntax [26] close to the C language. As most existing video decoders are provided in C, corresponding configuration information was easily derived from the decoders. However, the imperative form of the syntax limits their execution to a sequential process on a single processing unit. One objective of this paper is thus to increase the abstraction level of configuration information to suit the widest range of heterogeneous systems.

As such, dataflow representation of application is a way to tackle the constraint of serial execution of programs that is commonly found in most of programming languages. A dataflow representation, as depicted in Fig. 2, is modeled as a directed dataflow graph that represents data flowing between operations. The vertices are the operations of the application and edges are the data dependencies between these operations. The strength of a dataflow graph lies in the strong encapsulation of the operations of an application. Unlike most programming languages, dataflow programming focuses on algorithms instead of instructions. The execution of operations, closely related to functional programming, can be done concurrently and independently and are only driven by data.

2.1 MPEG Reconfigurable Media Coding: a standardized model for dataflow descriptions

Dataflow representation of decoders has been approved by the MPEG Reconfigurable Media Coding (RMC) consortium as a solution for deploying their existing and future standards [32]. This standard is composed of the Reconfigurable Video Codec (RVC) [33], dedicated to the specification of video applications, and Reconfigurable Graphics Codec (RGC) [31], dedicated to the specification of graphic applications (primarily 3D mesh decoders). The RMC framework supplies a normative standard library of multimedia coding tools, the Video Tools Library (VTL) for RVC and the Graphics Tool Library (GTL) for RGC. It also provides a set of decoder configurations to process graphic and video content expressed as networks of coding tools.

The dataflow model standardized in MPEG RMC is based on the Dataflow Process Network (DPN) [30]. This model is selected since it is the most expressive model among other dataflow models, such as KPN [14], SDF [28] or CSDF [8]. Moreover, DPN does not need the use of synchronization primitives, such as mutexes and semaphores. The operations of a DPN in a network are instances of Dynamic DataFlow (DDF) *actors* and edges are unidirectional FIFO channels. The execution of a DPN is broken into a sequence of actor executions called *actor firings* and the sequence of actor firings is determined by *firing rules*.

```
actor Abs () int I ==> uint O :
        pos: action I: [u] ==> O:[u]
             end

        neg: action I :[u] ==> O:[-u]
                guard u < 0
             end

        priority
                neg > pos;
        end

end
```

Fig. 3 CAL Actor Language description of an actor *Abs* that computes *Absolute Value* of tokens from its input I to its outputs O.

MPEG RMC defines actors, the coding tools in the VTL and GTL, with a domain-specific language called RVC-CAL [5,15]. RVC-CAL is tailor made for coding/decoding algorithm specifications that have inputs and outputs, states, and parameters. An actor is entirely encapsulated and communicates with other actors by sending and receiving tokens (atomic pieces of data) through communication ports, which represent the input and the output of the coding/decoding algorithm. Figure 3 gives an example of an actor RVC-CAL named *Abs* that computes the absolute value of a token received on the input port I and outputs on the output port O. This example produces and consumes a single token at each firing. However, RVC-CAL actors place no restriction on the number of tokens sent or received at each firing, which makes the model Turing complete [30].

RVC-CAL describes one actor firing with several *actions*. An action is the entry point of an actor, it may read tokens from input ports, compute data, change the state of the actor, and write tokens to output ports. The

body of an action is executed as an imperative function with local variables and imperative statements. In the given example, the action *pos* computes values received on port I as a variable u and sends this variable to the port O. In this particular special action, the variable u is only copied from the input I to the output. Conversely, the action *neg* copies a negative value of token from the port I to the port O.

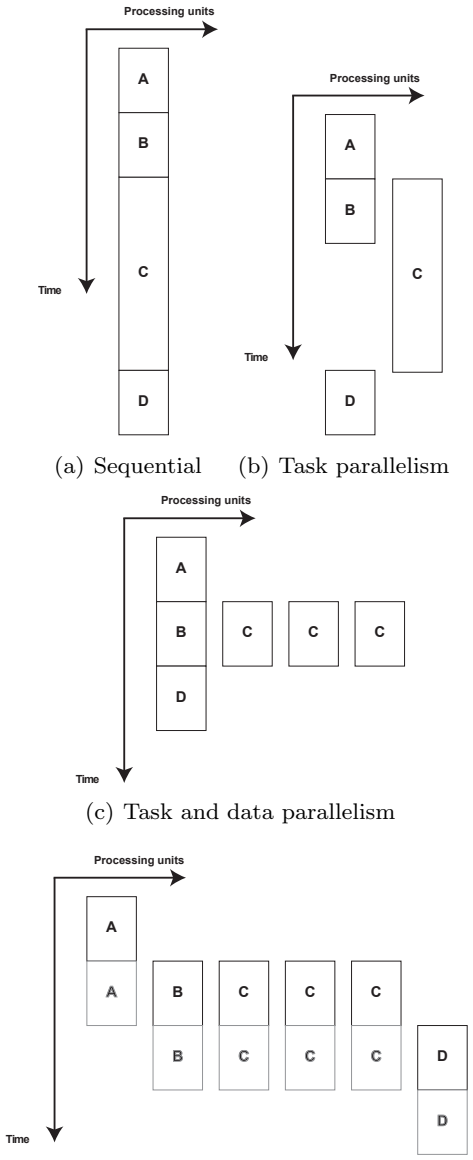
When an actor fires, a single action is selected. The selection is made according to the number and the values of tokens available on the input ports of the actor, and also according to its current state. One way to constrain the action selection is to associate a *guard* condition to one action. The guard condition allows a specification of additional firing conditions in an actor, where the firing rules of an action depend on the values of input tokens or the current state of the actor. In our example, the action *neg* may only fire if a value on port I is strictly negative. Actions selection may be further constrained by using a *Finite State Machine* (FSM) in the actor and/or *priorities* ($>$) between actions. Both conditions impose a partial order among the action firing. The use of priorities is illustrated on actor *Abs*; a negative value received on port I favors the firing of action *neg* against the action *pos*. Finally, an actor can contain parameters that are specified when this actor is *instantiated* in a network that references it. The reader may refer to [15] to obtain a precise description of the RVC-CAL paradigm.

2.2 Parallelisms stated in dataflow programs

Dataflow formalism has been extensively considered in the literature in order to produce efficient distributed models of execution for heterogeneous platforms [30, 16, 1, 3]. With its construction in the form of algorithms without any low-level details, a dataflow application states a large number of potential parallelisms that require adaptation according to the number of processing units of a particular execution platform.

Figure 4 illustrates these different degrees of parallelism. Assume that each actor of Figure 2 has computing time of $C_A = C_B = C_C/3 = C_D$ for each firing, so that 3 units of C are required for each unit of A, B, D. The DPN model states three degrees of parallelism (task, data and pipeline) applied to different granularities of description :

1. **Task parallelism** refers to disjoint algorithms on actor with no precedence relation. For instance, in Fig. 2, A precede B implies that B cannot be executed before A. On the other hand, B and C have no precedence relation, they therefore imply a task parallelism.
2. **Data parallelism** can be applied to actors with no state dependencies and no token dependencies between several successive firings. A set of data can thus be processed concurrently by duplicating this actor, so that it can consume N tokens at each firing on M processing units at the same time and with no overhead.



(a) Sequential (b) Task parallelism

(c) Task and data parallelism

(d) Pipeline, task and data parallelism

Fig. 4 Parallelizing a dataflow program from (a) a sequential execution on a single processing unit. (b) enhances (a) with one task parallelism between (B, C) using two processing units. (c) enhances (b) with data parallelisms on C using four processing units. (d) enhances (c) with pipeling A, B, C, D on six processing units.

3. **Pipelining** refers to the area of an application structured as a chain of actors. Each actor carries out an asynchronous and atomic process. Each pulls on a token from its inputs and push the processing results to its outputs and starts the same process again at the next clock. Pipelining does not enhance the throughput on one calculation, but on the processing of set of a data.
4. **Coarse-grain and fine-grain parallelisms** deal with the division granularity of the application's algorithms into actors. A fine-grain description is composed of small and atomic actors that frequently exchange tokens during processing. Conversely, a coarse-grain description is composed of computation intensive-actors, which exchange a large amount of tokens in one firing.

Data parallelisms are well suited for highly parallel architectures of processor such as DSPs and GPUs since actors can compute a large set of data at each clock cycle. Conversely, task parallelisms are well suited for systems that contain a massive resources of computation units, such as digital electronic systems. A fine-grain description of actors in a dataflow graph states more degree of parallelism, but also has a cost tied to the communication synchronization between actors. Thus, they are harder to execute efficiently on platform with reduced number of processing units, since several concurrent actors have to be executed sequentially. In this case, the execution order of actors is determined more or less efficiently with *scheduling strategies* [29,11,44].

3 Efficient and modular use of dataflow representations for multimedia applications

Many frameworks are available to produce static implementations of dataflow representation for heterogeneous platforms, the most famous being Ptolemy [10] and StreamIt [48]. Among them, the Orcc [53] compilation allows a static compilation of RVC-CAL actors for software and hardware platforms [53].

However, the use of dataflow programs is a relatively new field in the context of adaptive decoding, and, generally speaking, in dynamic compilation. This approach offers several benefits since one unified description form can be sent "as is" to target a large number of heterogeneous platforms, and the strong encapsulation of the operations favors the reuse of algorithms between different decoders.

This section aims to give details of these benefits. It also discusses the state-of-the-art on the current limitations of this approach for its practical application.

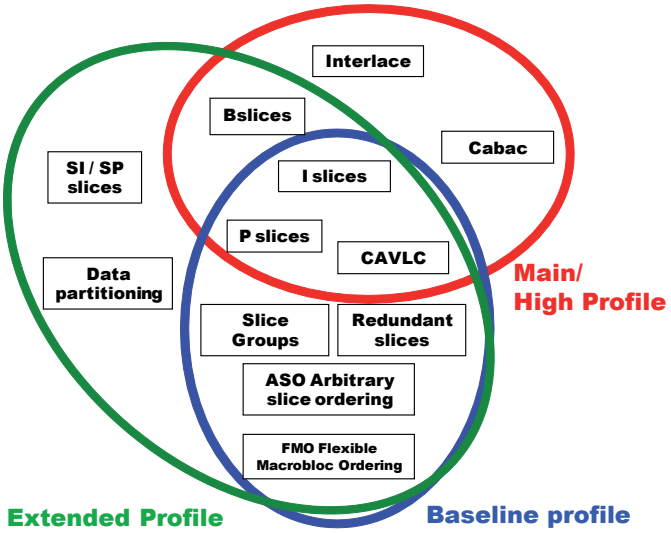


Fig. 5 Example of potential reusability on algorithms from the Baseline/Main/High/Extended profiles coming from the MPEG 4 *Advanced Video Coding*.

3.1 Partial recompilation of applications

Most of programming languages include code reusability, the ability of a code to be reused, for upcoming applications. However, the efficiency of code reuse highly depends on the permissiveness of one of the paradigms, which usually allows side-effects and dependencies with other codes in the same application.

In the case of dataflow programming, the strong encapsulation of operations allows the programmer to focus on the reusability of the algorithms. Dataflow representations of applications are natively modular and they facilitate the reconfiguration of algorithms by only modifying the topology of the graph from a dataflow network.

The ability to reuse code is particularly useful when dealing with multimedia decoders. Generally speaking, multimedia standards define many algorithms, or coding/decoding mechanisms, for different goals and requirements. Since they are implemented on many different devices with different use scenarios, they are structured in a set of *profiles* that are a subset of all the algorithms they define. Fig. 5 illustrates the pool of profiles of coding tools standardized in MPEG-4 part 10 Advanced Video Coding (AVC). This pool guarantees that at least 30% of the coding tools can be reused when switching from one profile to another [18]. The reuse percentage rises when switching to MPEG-4 AVC scalable profiles (SVC) [40] and MultiView profiles (MVC) [51].

While reusability is beneficial for programmers designing static and fixed application, dynamic applica-

tions, such as adaptive decoders, could also benefits from this particular property in dataflow representations. Indeed, the drawback of most dynamic compilers lies in the latency induced by the compilation of an application into enforceable machine code. This dynamic compilation avoids the application immediately starting its execution. Identifying exactly the machine code that can be reused when switching to a new decoder configuration would significantly reduce the compilation latency. The first configuration optimization, outlined in Section 4, is based on this observation. We propose an algorithm to dynamically identify actors and interchange them on-the-fly when switching from one dataflow representation of an application to another.

3.2 Scalable execution on single to multi-core platforms

The strong encapsulation of actors in dataflow representation also provides an explicit model of execution based on concurrent computing. Indeed, a dataflow representation can be executed by (1) executing in a separate concurrent process – or thread – each operation represented by the vertices of the graph (2) following the data dependency rules stated on the edge of the graph with message-passing communications. The set of concurrent processes in the network can be executed separately on platforms that contain a larger number of processing units than the number of actors in the application. The set of concurrent processes can also be executed on a single processor by interleaving the execution steps of each by a time slicing way. As such, dataflow representations match hardware platform, as the number of processing resources on-board can be greater than the number of operations in a network. However, given the restricted processing resources of software platforms, a genuine concurrency of the execution of dozens of processes can rarely be efficiently achieved because of context switching between each execution slice caused by the OS [30].

Instead of the context switching found in typical concurrent execution models, the DPN model has a special feature to allow a continuous execution of the operations of a graph. One process can sequentially test the firing rules from several actors, and fire an actor if a firing rule is valid. An efficient scheduling for dataflow programs consists in finding a, pre-defined or not, order of actor firings throughout the execution process capable of maximizing the use of all the processing units in one platform. Since actors in a DPN may have data-dependent behaviors, and the data are unknown in the system, the scheduling can be only done in the general case at run time. Lee and Parks [30] introduce a wide

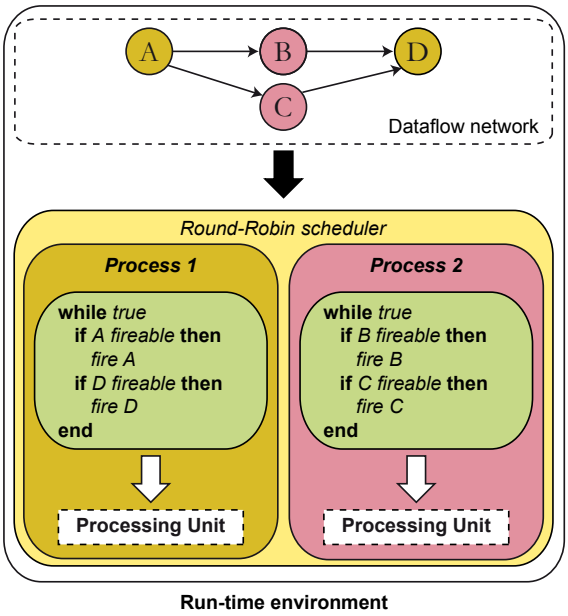


Fig. 6 Scheduling a dataflow network with a round-Robin strategy. 2 processes are assigned to a separate processing units and each process is assigned to two operations (respectively A, D and B, C). One process successively tests *fireability* and fires *fireable* the operations if the case arises.

variety of execution models – called *scheduler* – to optimize the run-time scheduling of actors. This variety is due to the fact that DPNs do not over specify application algorithms the way non-declarative semantics do.

A commonly-used execution model for DPNs consists of testing the validity of the firing rules for each actor in a DPN with a round-robin strategy. This strategy can be carried out, as illustrated in Fig. 6, by one process or several processes. On one hand, the round-robin scheduler has low complexity and it is simple to implement on the other hand, it may involve a low chance of success between the test and the validation of firing rules as this strategy makes no assumption on the topology of a graph and the singularity of its operations. For instance, in the example of Fig. 6, A may need to fire 10 times before validating firing rules of B and C. By using a round-robin strategy, the actual firings of A would be done one-by-one after a circular test of B, C, D in the graph. Furthermore, the firing rules of operation D may go through several testing phases while its activation only depends on operations B and C. As such, round-robin strategy may usually not be desirable if the fireability of operations is not symmetric. Data-driven, demand-driven or mixed data-driven/demand-driven strategies add more visibility to the topology of a graph [50]. They use the last operations fired and test the fireability of the next opera-

tions in the graph in the case of data-driven strategy, the previous operation in the graph for demand-driven strategy, or both previous and next operations in a mixed strategy. All these strategies may improve the chance of success between test and validation of firing rules in a graph, but also induce significant complexity in the scheduling process. For instance, the mixed data-driven/demand-driven strategy detailed in [54] is 5 more efficient compared to the round-robin strategy, but it also increases the run-time scheduling overhead with up to 10 more firing rules tested for a single actor.

The optimization detailed in Section 5 starts from the observation that many algorithms in standardized video coding tools consume a fixed amount of data to produce a fixed amount of data. This is, for instance, the case in the Discrete Cosine Transform (DCT) [17], common to several MPEG video standards, which takes a block of DCT coefficients – of size 8×8 in MPEG-4 part 2 and MPEG-2 – to recover a pixel-based block of the same size via several matrix computations. We propose in Section 5 to maximize the scheduling order on these operations without using any run-time scheduling strategies. The developed algorithm suppresses the overhead of runtime scheduling on these particular operations while staying compliant with conventional run-time scheduling on other operations.

4 Configuration optimization: dynamic compilation & partial recompilation of multimedia decoders

Dynamic compilation can be defined as the opposite of a static code compilation. Static compilation generates applications in the form of machine code, which match a specific software and hardware execution environment. The produced machine code is distributed as is and can be executed on machines that match the defined environment without any further modification.

Conversely, dynamic compilation requires a dynamic compiler to be included in the targeted environment. Applications are distributed in a higher-level form with a strong abstraction from any details of the execution environment. The dynamic compiler works at the receiver side to produce the enforceable machine code from one application. Dynamic compilers have been studied in literature and found to produce efficient compilations, such as Just-In-Time (JIT) [46] or Ahead-Of-Time (AOT) [23] compilation, which can significantly reduce delays to obtain the actual implementation of an application. A common use of dynamic compilation are virtual machines; they dynamically compile and execute a common intermediate representation of applications – known as *bytecode* – on a heterogeneous set of operating systems and processors.

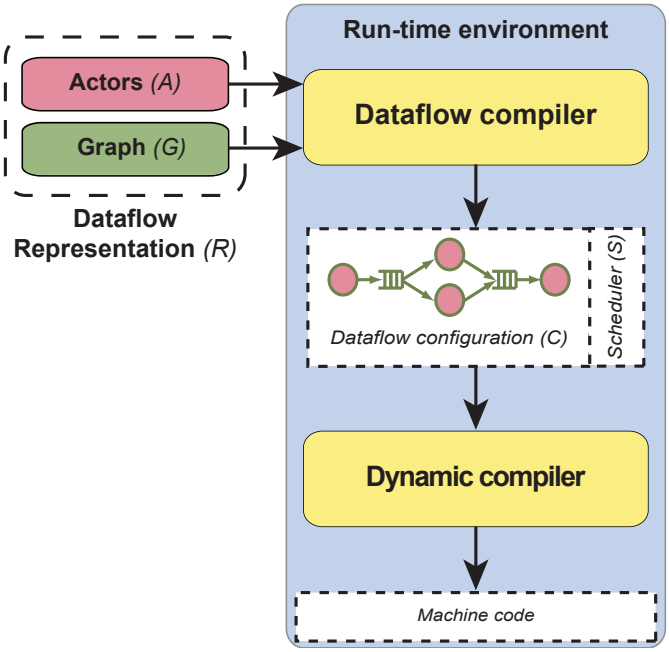


Fig. 7 Flowchart of the dynamic compilation process on a dataflow program.

The optimization detailed in this section takes the benefits of these mechanisms with the goal of supplying enforceable multimedia applications with a short delay. The resulting multimedia application preserves the strong encapsulation from the original dataflow representation up to the machine code in the execution environment. We propose the flowchart depicted in Figure 7 that extends the use of dynamic compilers to the support of the *configuration* and *reconfiguration* of dataflow programs. Following this structure, a reconfigurable implementation of dataflow representations becomes a three-step process:

1. A *dataflow compiler* acquires a dataflow representation (R), the *configuration information*, from an application in the form of one graph and several actors.
2. It produces a *configuration* (C) of the application in a single intermediate representation form and adds a model of execution, the *scheduler* (S), for its execution. This configuration may vary according to the run-time environment and the property of the underlying dynamic compiler.
3. The *dynamic compiler* receives the produced configuration and translates its intermediate representation into enforceable machine code that suit to the execution platform.

This section is organized in two contributions. The first contribution implements dataflow models to con-

serve their encapsulation up to the machine code of the run-time environment. We draw on usual denotational semantics from concurrent process networks to provide this automatic and conservative configuration of the dataflow representation. The second contribution makes it possible to adapt the generated configuration to a new dataflow representation.

4.1 Configuring a network of actors

The dataflow compiler in Fig. 7 produces a configuration of a dataflow representation by instantiating and connecting actors in a single representation of the corresponding application. As represented in Fig. 7, a description of an application is sent to the dataflow compiler as a dataflow representation $R = (A, G)$ where:

- A is a set of actors the application is depending on, and,
- G is a directed graph of vertices V, which reference actors in A, and edges E that represent the communication topology between V.

Following the DPN principle [30], the dataflow compiler produces from the directed graph $G = (V, E)$ a configuration $C = (I, D)$ of the application where:

- I is a set of *instances* of actors in A. An instance is a reflection – or a clone – of an actor a in A referenced by vertices V.
- D is a set of *directed*, point-to-point, order-preserving, and asynchronous FIFO connecting ports of instances. They correspond to the edge e in E.

The technical challenge when generating a configuration is to conserve the functional behavior of each actor along with its strong encapsulation.

We follow the *denotational semantic* used by Kahn [25] to describe the behavior of set of processes on communication channels. The processes are, for our case, instances in I and communication channels are FIFOs in D. In this notation, each FIFO carries a sequence of tokens at any time $X = [x_1, x_2, ...]$, where each x_i is a token. An empty FIFO channel that carries no tokens is an empty sequence denoted as $\perp$. A sequence X that precedes a sequence Y, e.g. $X = [x_1, x_2]$ and $Y = [x_1, x_2, x_3]$, is denoted $X \sqsubseteq Y$. The set of all possible sequences is denoted S, while S^p is the set of p-tuples of sequences on the p FIFO channels of a process. In other words, $[X_1, X_2, ..., X_p] \in S^p$ represents the sequence consumed/produced by a process. S^2 corresponds to $s_1 = [[x_1, x_2, x_3], \perp]$ or $s_2 = [[x_1], [x_2]]$. The length of a sequence is given by $|X|$; similarly the length of an element $s \in S^p$ is denoted $|s| = [|X_1|, |X_2|, ..., |X_p|]$. Thus, $|s_1| = [3, 0]$ and $|s_2| = [1, 1]$.

Based on this denotational semantic, Khan defines a process with m inputs and n outputs as a continuous and monotonic function:

$$F : S^m \rightarrow S^n \tag{1}$$

A process is triggered when S^m appears on it inputs; it is activated iteratively as long as S^m exists. Conversely, the process is suspended when S^m does not exist on its input. In other terms, reading from a FIFO can be blocking for one process until S^m appears again.

Dennis [13] extends the principle introduced by Khan with the notion of *firing rules* in processes. By using this notion, a process becomes an *actor* that can be triggered by several sequences S^m on its inputs; a network of actors becomes a DPN that does not require an environment of suspension/activation of process specific to the Kahn model. Lee [30] introduces a new denotational semantic for actors that can have N firing rules:

$$\mathrm{R} = [\mathbf{R}_1, \mathbf{R}_2, ..., \mathbf{R}_N] \tag{2}$$

A firing rule $\mathbf{R}_i$ defines a finite sequence of patterns, one for each input m of the actor:

$$\mathbf{R}_i = [P_{i,1}, P_{i,2}, ..., P_{i,m}] \in S^m \tag{3}$$

A pattern $P_{i,j}$ is an acceptable sequence of tokens in R_i on one input j from the input m of an actor. It is satisfied if and only if $P_{i,j} \sqsubseteq X_j$ where X_j is the sequence of tokens available on the j^{th} FIFO channel. The pattern $P_{i,j} = \perp$ designates any empty list where any available sequence on input j is acceptable. The pattern $P_{i,j} = [*]$ is acceptable for any sequence *containing at least one token*. The length of a pattern $P_{i,j}$ is denoted $|P_{i,j}|$.

An actor fires when at least one of its firing rules is satisfied. Lee and Parks [30] define the firing of an actor as *functional*, and the test of a set of firing rules of an actor as a *sequential* process. The term functional means that the output tokens resulting from an actor firing are purely a function of the input tokens. The term sequential indicates that firing rules can be tested in any pre-defined order. The role of the scheduler added in a configuration of a dataflow representation is therefore to test sequentially and continuously the validity of firing rules, as defined in Equations 2 and 3, and fire the corresponding function F as defined in Equation 1.

4.2 Configuring an actor

The test of the firing rules from each actor in a configuration requires evaluating the absence or presence of tokens from/to the FIFO channels. It also requires to "*peek*" at values from inputs to check if a sequence S^m matches the pattern of one of the firing rules $\mathbf{R}_i \in \mathbf{R}$ on the input of an actor. An actor consumes a finite number of tokens of value $|P_{i,j}|$ from its input j when firing rule $\mathbf{R}_i$ is satisfied. It produces a finite number of tokens on its output k of size $|s_k|$ at each firing. To conserve the strong encapsulation of each actor, we insert two functions in the configuration to evaluate firing rules according to the sequence of tokens include in the FIFOs:

1. *Has tokens function:* Gets the number of tokens available in a FIFO. It evaluates for one firing rule $\mathbf{R}_i$ if the sequence X_j of the FIFO j corresponds to the condition $|P_{i,j}| \leq |X_j|$
2. *Peek function:* "Peeks" at a fixed number of tokens from a FIFO without consuming. It evaluates in a firing rule $\mathbf{R}_i$ if the sequence X_j of the FIFO j corresponds to the condition $P_{i,j} \sqsubseteq X_j$.

The monotonic property of firing functions allows a configuration to work on a bounded FIFO. Nevertheless, this property also requires that there is enough room in the FIFO to store the sequence S^n produced by a firing function. We insert three functions in the configuration to handle the firing of an actor:

1. *Has rooms function:* Get amount of room available in a FIFO. It evaluates if a sequence of token $s \in S^m$ can be store in an FIFO of size l that already contains a sequence X; in other terms it evaluates $|s| \leq |X| - |l|$.
2. *Read function:* Consumes a sequence of tokens X from a FIFO.
3. *Write function:* Writes a sequence of tokens X to a FIFO.

An RVC-CAL actor, depicted in Fig. 3, can be considered a special case of actor from DPNs. It encompasses a persistent state Σ that contains at any time a set of p values $[v_1, ...v_p]$. These states are also initialized to the set of values $\sigma^0 = [v_1^0, ...v_p^0]$ at the instantiation of the actor. Moreover, one *action* i of an RVC-CAL actor can be considered a single firing function $f_i \in F$ such as:

$$\mathrm{F} = [f_1, f_2, ..., f_N] \tag{4}$$

where each function $f_i \in F$ is associated with one firing rule $\mathbf{R}_i \in \mathbf{R}$.

The firing functions can have a localized side effect on the actor state Σ, therefore we can define their behavior as:

$$f_i : \Sigma \times S^m \rightarrow \Sigma \times S^n \tag{5}$$

The associated firing rule can also depend on a value Σ_i from the actor state Σ, namely,

$$\mathbf{R}_i = [P_{i,1}, P_{i,2}, ..., P_{i,m}] \in S^m,\ \Sigma_i \in \Sigma \tag{6}$$

For example, the actor *Abs*, whose source code is given on Fig. 3, is specific case of an RVC-CAL actor with no persistent state ($\Sigma = \emptyset$). Its actions *pos* and *neg* can be depicted with two firing functions $F = [f_{pos}, f_{neg}]$ where:

$$f_{pos} : [s] \rightarrow [s] \tag{7}$$

$$f_{neg} : [s] \rightarrow [-s] \tag{8}$$

The firing functions (7) and (8) are respectively associated to firing rules:

$$\mathbf{R}_{pos} = \{\exists s \in S^m \mid s = [*]\} \tag{9}$$

$$\mathbf{R}_{neg} = \{\exists s \in S^m \mid s = [x_1],\ x_1 > 0\} \tag{10}$$

The inequality relation between the action *pos* and *neg* imposes an ordered sequence of tests on these firing rules, from the highest priority to the lowest, such as $R = \mathbf{R}_{neg} > \mathbf{R}_{pos}$. As such, one firing rule evaluating as valid invalidates all the firing rules of lower priority.

4.3 Adapting a configuration to a new dataflow representation

The previous section has identified the elements that composed a configuration of a dataflow representation and the mechanism for scheduling this configuration. The same configuration is then sent to the dynamic compiler as a two separate sets of instances and the directed FIFOs $C = (I, D)$. The dynamic compiler allocates space in the memory of the executing platform to store the state Σ from each instance in I and to store tokens in directed FIFOs from D. The set of firing rules, R, and the set of firing functions, F, in I are then compiled into a set of subroutines, for a later execution by the configuration scheduler.

An essential feature of dynamic compiler is that they conserve the location of every subroutine and memory allocated from the original source code in order to process their dynamic compilation. This location information makes it possible to modify subroutines and memory spaces on-the-fly in the run-time environment in order to switch the operations or the connections of

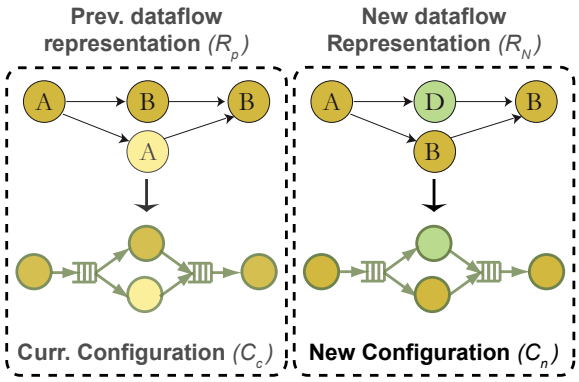


Fig. 8 Example of reconfiguration from one dataflow representation (R_p) to a new dataflow representation (R_n). 1 instance of A is removed (yellow) in the configuration; 1 instance of actor D is added (green); 2 instances of actor B and 1 from actor A are reused (orange).

a dataflow representation or to apply load balancing strategies in the scheduler. We call this process a *run-time adaptation* by applying a *partial recompilation* on dataflow programs.

Besseron [4] defines the three basic functions of a run-time adaptation: monitoring, making decisions and reconfiguring dataflow programs. Monitoring consists of knowing the context of a program before the adaptation. Reconfiguration is the actual modification from the former configuration to set a new configuration. The decision is therefore the understanding of the reconfiguration, it chooses an appropriate reconfiguration according to the old and the new context.

There are two types of reconfiguration approaches: dynamic and static reconfiguration [4]. Static reconfiguration takes place before or at the beginning of the execution of an application. Monitoring is then reduced to the machine code and the memory space currently used in the program, regardless of the states in the program. Conversely, dynamic reconfiguration can be performed several times during the execution of the application, it monitors the machine code of the program and its actual state, and it identifies *quiescent points* [34] in the application in order to not interfere with its current execution.

Traditional approaches in adaptive decoders [43] reduce the complexity of the run-time adaption by adding decisions in the configuration information. However, these decisions bind the adaption of an application to a dynamic reconfiguration where the transmitter and the receiver of configuration information know the context of the application at any time. This case can only occur when the decoding application processes similarly coded content during all the dynamic reconfiguration process.

A common use case of context changing is when the run-time environment requires processing new coded bitstream without informing the transmitter. We propose sending a complete dataflow representation when an adaptation of one application is required, to place the complexity of the monitoring and the decision at the receiver side of the adaptive decoding context. Thus, the strong encapsulation of the actors makes it possible to intuitively identify the operation that are wasteful or lacking in the new dataflow representation. For instance, Figure 8 depicts a reconfiguration of a dataflow representation R_p composed of 2 instances of an actor A and 2 instances of an actor B. A new dataflow representation R_N is received on the run-time environment, composed of 1 instance of actor A, 2 instances of actor B and one instance of new actor D. The new configuration C_n from R_N can reuse 2 instances of B and 1 instance of A from the current configuration C_c, which has been generated from R_p.

4.4 Monitoring, decisions, and reconfiguration on configurations

At the monitoring side, the current configuration C_c owns at any time a set of instance I_c and a set of directed FIFO D_c. I_c and D_c are representations of the previously-transmitted dataflow representation R_p, composed of a graph $G_p = (V_p, E_p)$ and of a set of actors A_p. Each instance i_c in I_c owns a current state Σ_c in Σ. Each directed FIFO d_c in D_c owns a sequence of token X. Before starting the execution of the configuration, the entire instances it comprises are in their initial states $\sigma^0 \in \Sigma$. The directed FIFOs contain an empty sequence of token $X = [\perp]$.

An adaptation is required on the configuration when a new dataflow representation, R_n, composed of a graph $G_n = (V_n, E_n)$ and a set of actors A_n, is sent to the receiver. As the configuration is only compose of strongly encapsulated components, the dataflow compiler can apply the following decisions:

1. Each instance i_c in I_c that refers to an actor $a \in A_p$ and where $a \notin A_n$ must be removed from the configuration. In other terms, all the instances from I_c that refer to an a that is in the relative complement $A_p \setminus A_n$ are marked with a "removed" decision. We denote this decision as $\ominus$.
2. Each vertex v_n in V_n that refers to an actor $a \in A_n \setminus A_p$ must be compiled as a new instance in I_c. We denote this decision as $\oplus$.
3. One occurrence v_n in V_n that refers to an actor $a \in A_n \cap A_p$ must be linked to one instance i_c in I_c that

refers to the same actor a in A_p. We denote this decision as $\odot$.
4. Each vertex v_n in V_n where there is no remaining occurrence i_c in I_c are marked by the decision $\oplus$.
5. The remaining instances in I_c – which have not being marked by a decision– are marked by the "removing" decision $\ominus$.

At the reconfiguration side, instances from I_c marked as $\ominus$ have their machine code from its firing functions F and its firing rules R cleared in the run-time environment, along with the memory space allocated to store their state Σ. Reconfiguration of the vertices in V_n marked as $\oplus$ includes generating the machine code and allocating the memory for F, R and Σ according to the actors they reference. The reconfiguration process also takes into account the scheduler associated with the configuration where decisions $\ominus$ and $\oplus$ add or remove the tests and the firings executing on the corresponding instance.

To produce a static reconfiguration, the directed FIFOs from D_c must be set again to the empty sequence $X = [\perp]$ and must be reconnected to a port of an instance in the new configuration. The number of FIFOs must match the number of edges in E_p. The states Σ from each instance marked as $\odot$ in I_c are set to their initial states σ^0.

For a dynamic reconfiguration, a directed FIFO from D_c, which connects two instances (x, y), where x or y is marked as $\ominus$, is cleared from the memory space of the run-time environment. The edges from E_n, which connect two instances (x, y), where x or y is marked as $\oplus$, are converted into new FIFOs in the configuration. Reconfiguring an instance marked as $\ominus$ must be performed during its quiescent point, i.e. when the instance is not consuming or producing tokens from or to FIFOs in the configuration. Reconfiguring a instance cannot occur during its firing period.

One crucial requirement of the algorithm is that all actors with one same behavior must own an equivalent identifier from the previous to the new reconfiguration in order to process their matching. This condition fits the MPEG RMC framework, as all coding tools from the VTL and GTL are identified with a unique string.

5 Execution optimization : efficient clustering of algorithms on processing units

In the previous sections, we stated that one essential benefit of the DPN model lies in its strong expressive power, so as to simplify algorithm implementation for programmers. This expressive power includes: the ability to describe data-dependent computations through token production/consumption, where production/consumption may vary according to values of tokens; the ability to express non-determinism, which can be used to construct actors that respond to unpredictable sequences of tokens; and, the ability to produce time-dependent behaviors that rely on the time at which tokens are available on the input of an instance.

However, when dealing with the scalability of this model, we stated that this strong expressive power incurs a cost on the efficiency of its implementation, as several operations may be scheduled at run-time on a single processing unit. The overhead caused by a scheduling strategy, along with its variable chance of success between test/validation of a firing rule for each operation, can create a succession of synchronization issues between the firing of instances in a configuration. This issue can ultimately lead to inefficient implementation of dataflow programs or to unsteady performance on their executions. The granularity of an application, i.e., the number of actors to schedule in the configuration, becomes an important factor that can prevent synchronization issue of instances.

The challenge when optimizing the execution of a configuration is then to conserve the strong expressive power of DPN while reducing the overhead caused by its required run-time scheduling. In this section, we propose a process that reduces the number of actors that are required to be scheduled at run-time, by clustering network regions that have a locally static behavior. We mean by one locally static region a set of connected instances in the configuration that has a firing order we can determine statically, regardless the data stored in the FIFOs of the configuration. The contribution is based on three existing algorithms; we apply them to produce the following process:

1. We detect instances with predictable behaviors in a dataflow representation by using *classification algorithms* [52,55],
2. Predictable instances connected amongst themselves are clustered into a single node if they match the *composition theorem* [41]. The resulting cluster becomes a *composite node* in the graph of the dataflow configuration,
3. Instances grouped in a composite node are scheduled at compile time in the configuration with a *Single-Appearance Scheduling* (SAS) [36]. The other remaining instances, along with the resulting composite nodes, are scheduled at run-time.

We provide details of each of these processes in the following sections.

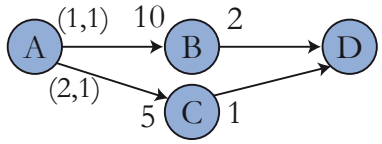


Fig. 9 Dataflow Process Network where actors B and C have a *Synchronous DataFlow* (SDF) MoC and actor B has a *Cyclo-Static Dataflow* (CSDF) MoC.

5.1 Detecting predictable actors

The literature introduces many algorithms, such as [52] and [55], to classify dynamic actors into restricted Model of Computations (MoCs). Restricted MoCs represent different trade-offs between expressiveness, in exchange for considerable advantages such as compile-time predictability [30]. Compile-time predictability includes the ability to determine, partially or in its entirety, the firing sequence of a composite node only composed of predictable actors.

The subsets of classes of actors, for which the firing sequence can be entirely determined at compile time, are called Synchronous DataFlow (SDF) [28] and Cyclo-Static Dataflow (CSDF) [8]. An SDF actor consumes and produces a constant number of tokens at each firing. It may have a single firing rule, which is valid for any sequence S^m of a certain size on its inputs [29,28]. In the case where an actor has several firing rules, an actor is SDF if all its firing rules have the same consumption, which mean for $\mathbf{R}_A \in R$ and $\forall \mathbf{R}_B \in R$:

$$|\mathbf{R}_A| = |\mathbf{R}_B| \tag{11}$$

All the firing functions of an SDF actor must also produce a fixed number of tokens at each firing, which means for $f_a \in F$ and $\forall f_b \in F$:

$$|f_a(s)| = |f_b(s)| \tag{12}$$

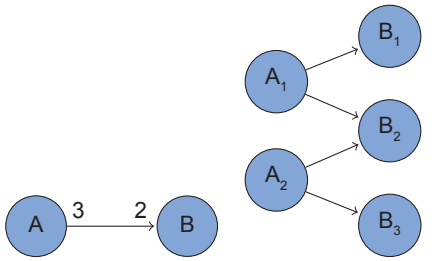
for any $s \in S^m$ and $s_b \in S^m$

The CSDF MoC [8] extends SDF actors by allowing the number of tokens produced and consumed to vary cyclically. This variation is modeled with a state in the actor, which returns to its initial value after one period. Figure 9 depicts a mixed implementation of dataflow actors where the instance A is in keeping with the CSDF MoC, B and C are in keeping with the SDF MoC, and D is dynamic. Each edge is annotated by the consumption/production rate known at each firing. An annotation of "5" on the head of an edge indicates that the instance C is SDF and consumes five tokens at each firing. A production $(2, 1)$ on the tail of an edge indicates a pattern of production on the CSDF instance A where its firing produces alternatively one and three tokens.

Classification algorithms in [52,55] analyze the behaviors of an actor in reaction to sequence of tokens S^m, where the size $|S^m|$ is increasing. In the case where the production and the consumption of the actor follows a fixed pattern or cycle pattern, the actor is detected as predictable; it follows the rules from one of the presented MoC. These algorithms involve a resource consuming computation as every actor in a network must be tested on a wide series of token. However, in an adaptive context, this process can be performed at the transmitter side in order to avoid any overhead in the run-time environment of the receiver.

5.2 Clustering predictable actors in one composite node

The objective of clustering several actors into a single node, the composite node, is to obtain a valid sequence of firing in it that can be determined before its actual execution in the configuration. This valid sequence avoids the use of any run-time scheduling for its execution. As such, an essential condition to set a composite node is to determine whether such a sequence of firing is possible. We call this condition *consistency checking* or *clock calculation* on the instances from a composite node.



(a) Strongly connected SDF actors (b) Precedence graph

Fig. 10 Illustration of (a) two strongly connected SDF actors (b) with their corresponding precedence graph.

If such a sequence exists, it must then be determined according to the precedence of the actors in the composite node. This step corresponds to the creation of a *static scheduling* strategy. An essential hypothesis to prove the existence of static scheduling concerns the sequence of tokens owned by the FIFOs in the configuration. A valid sequence of firing ensures that there will be neither accumulation nor lack of tokens – called a *deadlock* – between each firing of predictable actors. A common practice to check this hypothesis is to build a precedence graph, depicted in Fig. 10, where the

consumption/production on each edge is homogeneous and equal to one. Pino states in [41] that a precedence graph, which contains no cycle, is a Directed Acyclic Graph (DAG). This DAG ensures that there exists at least one possible static scheduling strategy to be applied in the composite node. In other terms, there exists a sequence of firings in a DAG that can produce a continuous series of tokens in FIFOs where these FIFOs return to an initial and determined state at the end of a cycle.

Pino also introduces in [41] three *composition rules* that validate the clustering of two SDF actors in a single composite node. He uses this clustering algorithm as a pre-pass to define static scheduling strategies that reduce the number of vertices in a dataflow representation only composed of SDF actors. The scheduling strategy such defined minimizes the synchronization overhead on multi-threaded implementations and maximizes the throughput by grouping buffers. The composition rules help ensure that the resulting precedence graph will always be a DAG and that the static execution of the composite node may not cause any deadlock. We depicted these three rules on two SDF instances (x, y) in Fig. 11 and we extend them to the general use of dataflow actors, with the help of the classification algorithms. We can observe that the *composition rules* include the notion of delays, denoted δ. Delays are initialization tokens placed at the compilation of an SDF network. They are not yet included in classification algorithms.

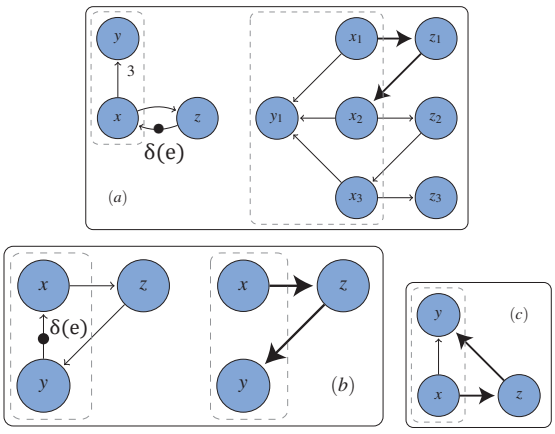


Fig. 11 Composition rules on two strong connected SDF instances (x, y) (left) with their impact on the corresponding precedence graph (right): (a) illustrates the violation of the first precedence shift condition, (b) illustrates the violation of the hidden delay condition, and (c) illustrates the violation of the cycle introduction condition. Each violation introduces a cycle in the precedence graph denoted with bold arrows.

To process the proposed configuration optimization, we remove the notion of delay from the composition rules and we deduce two new conditions in the composition rules. These 2 rules ensure that the clustering of two instances will never cause any deadlocks in the application. Supposing that (x, y) are two instances that reference either a SDF or a CSDF actor:

1. If $p(x)$ and $c(y)$ are the number of tokens produced by x and consumed by y at each firing respectively, there exists a positive integer k such a $c(y) = k \times p(x)$,
2. There is no *simple* path from x to y which contains more than one arc. A simple path refers to a path that does not pass twice through the same arc of a graph, *i.e.* all of whose arcs are distinct.

The implementation of these conditions is made by passing through all the pairs (x, y) connected with at least one edges. We mark as candidate for clustering all the ones that have both a consumption $q(y)$ and a production $q(x)$ known. Each candidate (x, y) then passes through the following steps:

1. if $c(y) \neq k \times p(x)$, the pair (x, y) is removed from the candidate list;
2. a breadth-first search or a depth-first search is applied on the network graph to find all the simple path between x and y. If the pair (x, y) has more than one simple path, it is removed from the candidate list, and,
3. the couple (x, y) is finally clustered into a single composite node λ and this composite node λ is connected in the graph. The incoming edges from x become the incoming edges of λ, which is marked as having a consumption of $k \times c(x)$. The outgoing edges from y become the outgoing edges of λ and are marked with a production $p(y)$.

This process is iterative and is executed until there exist no candidate to cluster in the network. It results in a hierarchical configuration where the highest level is composed of both instances and composite nodes. This hierarchy is scheduled using a run-time strategy. The lower levels of the hierarchy are clusters of SDF and CSDF actors that can be scheduled at compile time. The firing rules of the composite of the higher hierarchy are the firing rules of the same instances connected in the lower hierarchy. These firing rules are only based on the presence of a number of tokens on the input of the composite node.

5.3 Static scheduling of actors in composite nodes

The objective of the static scheduling step is to determine the optimum firing order for all instances in the

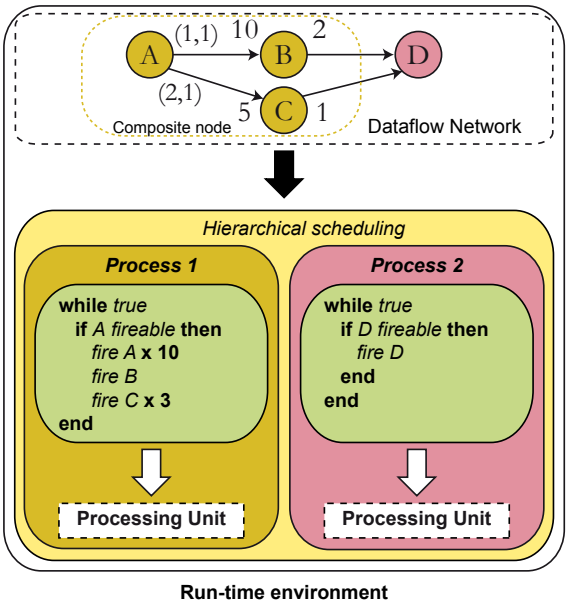


Fig. 12 Clustering execution of a configuration network where actor A, B and C have SDF MoCs and actor D is dynamic. The first process fires sequentially: 10 times A, one time B and, three times C, if A is detected fireable. Repetition factors ($\times$10and $\times$3) are implemented using *for loops*. The second process continuously tests fireability of a dynamic actor, D, and fires when D is fireable.

composite node. It exists a large number of scheduling strategies for static actors [6,7,47,49,37] based on different optimization criteria. In the case of an execution on a single processing unit, these criteria are structured around the minimization on the number of instructions to schedule a configuration or around the minimization of the memory allocated to produce the communication FIFOs in the run-time environment.

In an adaptive decoding context, the dynamic compiler has strong constraints on the size of code to be generated as it introduces a delay before the execution of the configuration. We thus favor the Single-Appearance Scheduling (SAS) [36] strategy, as it represents the optimum strategy for code minimization where all repetitions of a same instance can be found side by side. As represented in Fig. 12, the code generated from SAS can make an extensive use of *for* loops for each repetition factor.

SAS aims at determining the minimum periodic sequence of firing for an instance actors in a composite node, so that the execution of a configuration introduces no deadlock. This sequence is defined by a *repeat* vector$\mathbf{q} = (q_1, q_2, \cdots, q_n)$ where each q_i is the number of firing of the instance i in the composite node.

The deadlock-free calculation between two instances x and y on an arc $e = (x, y)$ must respect the following equality:

$$\pi(e)q(x) - \chi(e)q(y) = 0 \tag{13}$$

In order to incorporate all the pairs (x, y) from the composite node, this equation can be represented as a matrix of consumption/production called *matrix topology*. It is defined by $\boldsymbol{\Gamma} = (\gamma_{i,j})_{1\leq i\leq m, 1\leq j\leq n}$ for a composite node that consists of m instances and n connections where:

$$\gamma_{i,j} = \gamma^+_{i,j} - \gamma^-_{i,j} \tag{14}$$

with:

$$\gamma^+_{i,j} = \begin{cases} \pi(i) \text{ if } \mathsf{src}(i) = j \\ 0 \quad \text{otherwise} \end{cases} \quad \gamma^-_{i,j} = \begin{cases} \chi(i) \text{ if } \mathsf{dst}(i) = j \\ 0 \quad \text{otherwise} \end{cases} \tag{15}$$

The resulting matrix topology defines a set of *balance equations* $\mathbf{q} = \mathbf{0}$ with an infinite number of solutions. The minimal positive solution is a set of integers that defines the *Basis Repetition Vector*(BRV) $\mathbf{q_{BRV}} = (q_1, q_2, \cdots, q_n)$ for the m instances of a composite graph. Determining a BRV can take place with Gaussian elimination on the topology matrix as described in [39]. The final step is to define a chronological order to apply BRV on each instance. This topological order crosses over the graph of the composite node through its entry point and assigns an increasing weight to each instance crossed. The corresponding scheduler is finally generated from the lowest weight to the highest, with its corresponding BRV. As all consumptions and productions from the instances of a configuration are determined, the size of each FIFO can be determined. The resulting composite node is thus optimized in terms of scheduling and memory space compared to an implementation following the DPNs' rules introduced previously.

By way of example, the topology matrix from the composite node on Figure 12 is of size $m = 3$ and $n = 2$ is as follows:

$$\begin{pmatrix} (1,1) & -10 & 0 \\ (2,1) & 0 & -5 \end{pmatrix}$$

The resolution of this matrix by the method of Gaussian elimination gives a BRV q of size $m = 3$ where:

$$\mathbf{q} = \begin{pmatrix} 5 \ \mathbf{(\times 2)} \\ 1 \\ 5 \end{pmatrix}$$

This example lands on the specific case of CSDF, the first row of the topological matrix, which represents consumption/production of A, must be set to the

full period of firing, 2 and 3 for $(1,1)$ and $(2,1)$) respectively. The resulting BRV, for A the first line, must then be multiplied by the length of the firing period ($\times\mathbf{2}$).

The SAS strategy has been designed to be used on a single processing unit. However, algorithms exist that can map and schedule SDF graphs, which in our case can come from any composite nodes onto multiprocessors in linear time with respect to the number of instances and processors [39]. These algorithms can be used to extend the proposed clustering to specific processing units such as a DSP or GPU.

6 Applications and results

The two presented optimization algorithms were implemented in an open-source adaptive decoder called JADE, which is a part of the *Open RVC-CAL Compiler* (Orcc) project. This adaptive decoder is based on the Low-Level Virtual Machine (LLVM) [27] and can thus process a Just-In-Time compilation of configuration information on a wide range of platforms (X86-64, PowerPC, ARM...). All the results presented in this section are reproducible by downloading this project and the associated library of coding tools from *http://orcc.sf.net*. The reader can refer to [21] to find information about this adaptive decoder and to [20] to find information about the actor representation it uses.

The two optimizations are tested on two dataflow representations of decoders standardized in MPEG RVC. The first representation corresponds to the *Simple Profile* (SP) specification of the MPEG-4 part 2 standard and the second representation corresponds to *Constrained Baseline Profile* (CBP) specification of the MPEG-4 part 10 Advance Video Coding (AVC) standard. Details of these two dataflow representations are given in [18].

Two other dataflow representations of decoders are also used to valid the partial recompilation process. The first representation is a proprietary specification from Ericsson [42] of the MPEG-4 SP standard and the second representation is the MPEG RVC specification of the *Fidelity Range Extensions* (FREXt) profile from MPEG-4 part 10 AVC. All these four representations can be download from *http://orc-apps.sourceforge.net*.

Finally, the test sequences used are the validation sequences provided by the MPEG consortium. For MPEG-4 part 2 SP, it is the sequence *foreman* of size CIF (352×288) composed of 300 frames at 30 frame/sec. For MPEG-4 AVC CBP, it is sequence *combine* of size QCIF (176×144) composed of 1700 frames at 30 frame/sec. The testing results are taken from an Intel E6600 Core2 Duo processor at 2.40 GHz running on Windows XP.

Table 1 Gain of configuration optimization: a full dynamic compilation and a partial dynamic recompilation are applied when switching from/to standard (stand.) and proprietary (prop.) dataflow representations of a same MPEG-4 part 2 Simple Profile; and when switching between two profiles of the same standard MPEG-4 part 10 AVC, namely, the Constrained Baseline Profile (CBP) and Fidelity Range Extensions (FREXt) profile.

Compilation	Dynamic	Partial	Gain
MPEG part 2 Simple Profile			
Stand. to prop.	1188 ms	380 ms	3
Prop. to stand	1141 ms	375 ms	3
MPEG part 10 AVC profiles			
CBP to FREXt prof.	4734 ms	3343 ms	1.4
FREXt prof. to CBP	3313 ms	1610 ms	2

6.1 Optimizing configuration of dataflow representations with two reconfiguration use cases

Table 1 shows the practical impacts of using the configuration optimization when switching from a standardized to a proprietary implementation of the Simple Profile specification in MPEG-4 part 2; and when switching from one profile to another profile from the same MPEG-4 part 10 AVC standard.

The dataflow representation of the MPEG-4 part 2 Simple Profile standardized in MPEG RVC uses 31 actors, which are instantiated in 51 instances in the configuration network of the decoder. The proprietary dataflow representation of the same profile uses in its configuration 30 actors on 60 instances. Thirty-seven instances, which is approximately 60% of all the instances in both configurations, were reused by switching from one representation to the other.

The CBP profile from MPEG-4 part 10 AVC uses 56 actors on 105 instances in the configuration network. The FREXt profile uses 74 actors on 128 instances. Eighty-five instances, which is 66% of the FREXt configuration, were reused between the two configurations.

The gain factor introduced by the partial recompilation highlights that reconfiguration is more suitable for fine grain variations in the dataflow representation. Indeed, the weakest gain (1.4) comes from profile reconfiguration where the monolithic parsers (1/3 of the source lines of code of the overall decoder) are not reused between the CBP and the FREXt profile. Conversely, the proprietary and standard representation of MPEG-4 part 2 SP uses the same parser. This use scenario highlights the needs of supporting the Bitstream Syntax Description Language (BSDL) [12], also standardized

Table 2 Impact on decoding performance of clustering with Single-Appearance Scheduling and Round-Robin scheduling (RR + SAS) compared to an entire Round-Robin (RR) scheduling. The sequence used are *combine* (CIF) for MPEG-4 part 2 SP standard and *foreman* (QCIF) for MPEG-4 part 10 CBP standard.

Strategies	RR	RR + SAS	Gain
SP (RVC)	144 fps	180 fps	1.25
AVC (RVC)	50 fps	81.5 fps	1.63

in MPEG RMC, in order to describe the behavior of the parser. This language can significantly reduce the granularity of a parser description and thus, should give the opportunity to obtain a more fine grained reconfiguration on this particular actor.

6.2 Optimizing execution of dataflow representations for single process execution

Table 2 highlights the practical impact on decoding performance by using the proposed execution optimizations on several dataflow representations and with a single execution process. A round robin-strategy is applied as a run-time scheduling strategy on the entire dataflow representation in the first case. In the second case, a mixed round-robin strategy is used for the higher-level hierarchy of the clustered dataflow representation and SAS scheduling is applied inside the composite nodes.

Dataflow representations from MPEG-4 part 2 SP standard have a more coarse-grain description in actors. There exist only a single locally static region that corresponds to the Inverse Discrete Cosine Transformation (IDCT). Only eight instances of the IDCT were clustered into one single composite node. Despite these small clusters, SAS scheduling on this composite node shows an increase of performance of up to 25% compared to a round-robin strategy applied to the entire representation.

The dataflow representation from the MPEG-4 AVC CBP standard has a more fine-grain description of the decoder with as many twice instances in the configuration network. Thirty instances are clustered into composite nodes with a gain of up to 50% in decoder performance.

The time to process the clustering on both dataflow representations is minimal (300 ms) compared to the dynamic compilation process, which is 4 seconds in the worst case.

These results show that, generally speaking, clustering has a significant impact on performance, particularly for fine grain dataflow representations. However, this gain can be increased by (1) rewriting actors from dataflow representations to expose more static behaviors and (2) extending the clustering to a wider class of MoC, such as the Quasi-Static actors [9].

7 Related work

This article extends the work presented in [19], which gives the complete translation process to obtain a DPN implementation from an RVC-CAL actor. A concrete implementation of this process is given using the LLVM paradigm.

Wipliez et al. present a new compilation framework to automatically translate a dataflow representation into C/C++ [53]. This automatic translation has been extend in [45] to hardware platform by providing HDL representation of dataflow decoder. On the other hand, Richardson et al. implement a Fully Configurable Video Coding (FCVC) framework [43] that uses the Universal Video Decoder (UVD) to process the adaptive decoding. Our contribution is a merging of these two approaches. It provides an automatic translation of dataflow representations following the adaptive decoding process. This contribution is a direct application of the work initiated in MPEG RVC [33].

Gu et al. present a technique in [22] to recognize a set of Statically Schedulable Regions(SSRs) within a dynamic dataflow program. SSRs have sets of ports which are statically coupled, i.e. the production of an output port must matches the consumption of the input port(s) it is connected to. However, this techniques has never been used in an adaptive context.

Reconfiguration of dataflow program is widely used for DSP processors [34] and FPGA devices [2]. Our contribution differs from existing reconfiguring algorithms by proposing a deliberately simple mechanisms that consider the limited constraints of our decoder representation and the limited computing capability of embedded system.

Clustering is the obvious next step of the classification method introduced in [55]. This method gives the opportunity to apply a tradeoff between the efficiency of a static scheduling strategy [55] and the expressive power of a dynamic scheduling of DDF actors [38].

8 Conclusions

This paper proposes two optimization algorithms that assert dataflow representations as suitable representations for configuration information in an adaptive decoding context. This approach is verified with experiments on two decoders from the MPEG RMC standard. The first optimization reduces the time to configure a

new decoder with already compiled algorithms from a previous dataflow representation. The second optimization shows the ability of a dataflow program execution to be remodeled to enhance its execution according to the computing resource of a given platform.

These optimizations are intended to fit current and emerging architectures that will be incorporated in multimedia terminals. Indeed, dataflow representations are widely used across many application domains including Digital Signal Processing, Graphics Processing Unit, many-core and massively multi-core systems. In this sense, work is currently in progress to extend these optimizations to be used in GPU, DSP and Multi-Processor System on Chip (MPSoC) systems. We target detecting instances and composite node that have no state dependencies between successive firings to obtain efficient computations on large sets of data that fit highly parallel processors. Another aim of the reconfiguration optimization is to conceive multi-purpose systems for FPGAs and ASICs, as depicted in [35]. The overall approach of this paper is also not limited to video or graphic decoding applications. The separation between network and processing tools provides a useful abstraction to design any signal processing application. For instance, dataflow representations from MPEG RMC has already been extended to audio applications (Reconfigurable Audio Coding) and to cryptographic applications.

References

1. Amagbégnon, P., Besnard, L., Le Guernic, P.: Implementation of the data-flow synchronous language signal. In: ACM SIGPLAN Notices, 6, pp. 163–173. ACM (1995)
2. Athalye, A., Hong, S.: Mapping of partial reconfigurable data flows to Xilinx FPGAs. In: SOC Conference, 2005. Proceedings. IEEE International, pp. 111–112 (2005)
3. Atkinson, D., Griswold, W.: Implementation techniques for efficient data-flow analysis of large programs. In: Software Maintenance, 2001. Proceedings. IEEE International Conference on, pp. 52–61. IEEE (2001)
4. Besseron, X.: Fault tolerance and dynamic reconfiguration for large scale distributed applications. Ph.D. thesis, Institut National Polytechnique de Grenoble-INPG (2010)
5. Bhattacharyya, S., Eker, J., Janneck, J., Lucarz, C., Mattavelli, M., Raulet, M.: Overview of the MPEG reconfigurable video coding framework. Journal of Signal Processing Systems **63**(2), 251–263 (2011)
6. Bhattacharyya, S., Murthy, P., Lee, E.A.: Renesting Single Appearance Schedules to Minimize Buffer Memory. Tech. Rep. UCB/ERL M95/43, EECS Department, University of California, Berkeley (1995)
7. Bhattacharyya, S.S., Murthy, P.K., Lee, E.A.: APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. Journal of Design Automation for Embedded Systems. In: DSP Block Diagrams into Efficient Software Implementations, DAES, pp. 33–60 (1997)
8. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.: Cyclo-static data flow. In: icassp, pp. 3255–3258. IEEE (1995)
9. Boutellier, J., Lucarz, C., Lafond, S., Gomez, V., Mattavelli, M.: Quasi-static scheduling of CAL actor networks for reconfigurable video coding. Journal of Signal Processing Systems pp. 191–202 (2011)
10. Buck, J., Ha, S., Lee, E., Messerschmitt, D.: Ptolemy: A framework for simulating and prototyping heterogeneous systems. International Journal of Computer Simulation **4**, 155–182 (1994)
11. Buck, J., Lee, E.: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In: Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on, vol. 1, pp. 429–432. IEEE (1993)
12. De Schrijver, D., De Neve, W., De Wolf, K., De Sutter, R., Van de Walle, R.: An optimized MPEG-21 BSDL framework for the adaptation of scalable bitstreams. J. Vis. Comun. Image Represent. **18**(3), 217–239 (2007)
13. Dennis, J.B.: First version of a data flow procedure language. In: Proceedings of the Colloque sur la Programmation, *Lecture Notes in Computer Science*, vol. 19, pp. 362–376. Springer (1974)
14. Edwards, S.: Kahn Process Networks. Languages for Digital Embedded Systems pp. 189–195 (2000)
15. Eker, J., Janneck, J.: CAL Language Report. Tech. Rep. ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley (2003)
16. Engels, M., Bilson, G., Lauwereins, R., Peperstraete, J.: Cycle-static dataflow: model and implementation. In: Signals, Systems and Computers, 1994. 1994 Conference Record of the Twenty-Eighth Asilomar Conference on, vol. 1, pp. 503–507. IEEE (1994)
17. Fan, C.: Fast 2-dimensional 4× 4 forward integer transform implementation for H. 264/AVC. Circuits and Systems II: Express Briefs, IEEE Transactions on **53**(3), 174–177 (2006)
18. Gorin, J., Raulet, M., Cheng, Y., Lin, H., Siret, N., Sugimoto, K., Lee, G.: An RVC dataflow description of the AVC Constrained Baseline Profile decoder. In: Image Processing (ICIP), 2009 16th IEEE International Conference on, pp. 753–756. IEEE (2009)
19. Gorin, J., Raulet, M., Preteux, F.: MPEG Reconfigurable Video Coding: from specification to a reconfigurable implementation. In: (To appears) Special Issue on Reconfigurable Media Coding (2012)
20. Gorin, J., Wipliez, M., Preteux, F., Raulet, M.: A portable Video Tool Library for MPEG Reconfigurable Video Coding using LLVM representation. In: Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on, pp. 183–190. IEEE (2010)
21. Gorin, J., Wipliez, M., Raulet, M., Preteux, F.: An LLVM-based decoder for MPEG Reconfigurable Video Coding. In: IEEE Workshop on Signal Processing Systems (SiPS 2010), Washington, D.C., USA, pp. 281–286 (2008)
22. Gu, R., Janneck, J., Bhattacharyya, S., Raulet, M., Wipliez, M., Plishker, W.: Exploring the concurrency of an MPEG RVC decoder based on dataflow program analysis. IEEE Transactions on Circuits and Systems for Video Technology **19**(11) (2009)
23. Hong, S., Kim, J., Shin, J., Moon, S., Oh, H., Lee, J., Choi, H.: Java client ahead-of-time compiler for embedded systems. In: ACM SIGPLAN Notices, 7, pp. 63–72. ACM (2007)

24. Janneck, J., Mattavelli, M., Raulet, M., Wipliez, M.: Reconfigurable video coding: a stream programming approach to the specification of new video coding standards. In: Proceedings of the first annual ACM SIGMM conference on Multimedia systems, pp. 223–234. ACM (2010)
25. Kahn, G.: The semantics of a simple language for parallel programming. In: J.L. Rosenfeld (ed.) Information processing, pp. 471–475. North Holland, Amsterdam, Stockholm, Sweden (1974)
26. Kannangara, C., Philp, J., Richardson, I., Bystrom, M., de Frutos Lopez, M.: A Syntax for Defining, Communicating, and Implementing Video Decoder Function and Structure. Circuits and Systems for Video Technology, IEEE Transactions on **20**(9), 1176–1186 (2010)
27. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California (2004)
28. Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE **75**(9), 1235–1245 (1987)
29. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **36**(1), 24–35 (1987)
30. Lee, E.A., Parks, T.M.: Dataflow Process Networks. Proceedings of the IEEE **83**(5), 773–801 (1995)
31. Lee, S., Lim, T., Jang, E., Lee, J., Lee, S.: MPEG Reconfigurable Graphics Coding framework: Overview and applications. In: Visual Communications and Image Processing (VCIP), 2011 IEEE, pp. 1–4. IEEE (2011)
32. Lucarz, C., Mattavelli, M., Thomas-Kerr, J., Janneck, J.: Reconfigurable Media Coding: A new specification model for multimedia coders. In: Signal Processing Systems, 2007 IEEE Workshop on, pp. 481–486. IEEE (2007)
33. Mattavelli, M., Amer, I., Raulet, M.: The Reconfigurable Video Coding Standard [Standards in a Nutshell]. Signal Processing Magazine, IEEE **27**(3), 159 –167 (2010)
34. Neuendorffer, S., Lee, E.: Hierarchical reconfiguration of dataflow models. In: Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on, pp. 179–188. IEEE (2004)
35. Nezan, J., Siret, N., Wipliez, M., Palumbo, F., Raffo, L.: Multi-purpose systems: A novel dataflow-based generation and mapping strategy. In: Circuits and Systems (ISCAS), 2012 IEEE International Symposium on, pp. 3073–3076. IEEE (2012)
36. Oh, H., Dutt, N., Ha, S.: Single appearance schedule with dynamic loop count for minimum data buffer from synchronous dataflow graphs. In: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems, pp. 157–165. ACM (2005)
37. Oh, H., Dutt, N., Ha, S.: Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In: ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation, pp. 497–502 (2006)
38. Parks, T.: Bounded scheduling of process networks. Ph.D. thesis, University of California (1995)
39. Pelcat, M.: Rapid prototyping and dataflow-based code generation for the 3GPP LTE enodeb physical layer mapped onto multi-core DSPs. Ph.D. thesis, Ph. D. dissertation, Dissertation, INSA Rennes, 210 p (2010)
40. Pelcat, M., Blestel, M., Raulet, M.: From AVC decoder to SVC: Minor impact on a data flow graph description. In: Picture Coding Symposium (2007)
41. Pino, J., Lee, E., Bhattacharyya, S.: A hierarchical multiprocessor scheduling system for DSP applications. In: asilomar, p. 122. Published by the IEEE Computer Society (1995)
42. von Platen, C., Eker, J.: Efficient realization of a cal video decoder on a mobile terminal (position paper). In: Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on, pp. 176–181. IEEE (2008)
43. Richardson, I., Kannangara, S., Bystrom, M., Philp, J., De Frutos Lopez, M.: Implementing fully configurable video coding. In: Proceedings of the 16th IEEE international conference on Image processing, ICIP'09, pp. 765–768. IEEE Press, Piscataway, NJ, USA (2009)
44. Sarkar, V.: Partitioning and scheduling parallel programs for multiprocessors. MIT press (1989)
45. Siret, N., Wipliez, M., Nezan, J., Rhatay, A.: Hardware code generation from dataflow programs. In: Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on, pp. 113–120. IEEE (2010)
46. Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H., Nakatani, T.: Overview of the IBM Java just-in-time compiler. IBM systems Journal **39**(1), 175–193 (2000)
47. Sung, W., Kim, J., Ha, S.: Memory efficient software synthesis from dataflow graph. In: ISSS '98: Proceedings of the 11th international symposium on System synthesis, pp. 137–142 (1998)
48. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for streaming applications. In: Proceedings of the 11th International Conference on Compiler Construction, pp. 179–196. Springer-Verlag (2002)
49. Thies, W., Karczmarek, M., Sermulins, J., Rabbah, R., Amarasinghe, S.: Teleport messaging for distributed stream programs. In: PPoPP'05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 224–235 (2005)
50. Treleaven, P., Brownbridge, D., Hopkins, R.: Data-driven and demand-driven computer architecture. ACM Computing Surveys (CSUR) **14**(1), 93–143 (1982)
51. Vetro, A., Wegand, T., Sullivan, G.: Overview of the Stereo and Multiview Video Coding Extensions of the H. 264/AVC Standard. Proceedings of the IEEE (2011)
52. Wipliez, M., Raulet, M.: Classification of Dataflow Actors with Satisfiability and Abstract Interpretation. International Journal of Embedded and Real-Time Communication Systems (IJERTCS) **3**(1), 49–69 (2012)
53. Wipliez, M., Roquier, G., Nezan, J.: Software code generation for the RVC-CAL language. Journal of Signal Processing Systems pp. 203–213 (2011)
54. Yviquel, H., Casseau, E., Wipliez, M., Raulet, M.: Efficient multicore scheduling of dataflow process networks. In: Signal Processing Systems (SiPS), 2011 IEEE Workshop on, pp. 198 – 203. Liban (2011)
55. Zebelein, C., Falk, J., Haubelt, C., Teich, J.: Classification of General Data Flow Actors into Known Models of Computation. Proc. MEMOCODE, Anaheim, CA, USA pp. 119–128 (2008)